Analysis of Threading Libraries for High Performance Computing

Adrián Castelló
Universitat Jaume I
Castellón de la Plana, Spain
Email: adcastel@uji.es

Rafael Mayo Gual
Universitat Jaume I
Castellón de la Plana, Spain
Email: mayo@uji.es

Sangmin Seo
Ground X
Seoul, Korea
Email: seo.sangmin@gmail.com

Pavan Balaji
Argonne National Laboratory
Lemmont, USA
Email: balaji@anl.gov

Enrique S. Quintana-Ortí
Universitat Politècnica de València
València, Spain
Email: quintana@disca.upv.es

Antonio J. Peña
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
Email: antonio.pena@bsc.es

***Abstract*—With the appearance of multi-/many core machines, applications and runtime systems evolved in order to exploit the new on-node concurrency brought by new software paradigms. POSIX threads (Pthreads) was widely-adopted for that purpose and it remains as the most used threading solution in current hardware. Lightweight thread (LWT) libraries emerged as an alternative offering lighter mechanisms to tackle the massive concurrency of current hardware. In this paper, we analyze in detail the most representative threading libraries including Pthread- and LWT-based solutions. In addition, to examine the suitability of LWTs for different use cases, we develop a set of microbenchmarks consisting of OpenMP patterns commonly found in current parallel codes, and we compare the results using threading libraries and OpenMP implementations. Moreover, we study the semantics offered by threading libraries in order to expose the similarities among different LWT application programming interfaces and their advantages over Pthreads. This study exposes that LWT libraries outperform solutions based on operating system threads when tasks and nested parallelism are required.**

***Index Terms*—Lightweight Threads, OpenMP, GLT, POSIX Threads, Programming Models**

1. Introduction

In the past few years, the number of cores per processor has increased steadily, reaching impressive counts such as the 260 cores per socket in the Sunway TaihuLight supercomputer [1], which was ranked #1 for the first time in the June 2016 TOP500 list [2]. This trend indicates that upcoming exascale systems may well feature a large number of cores. Therefore, future applications will have to accommodate this massive concurrency by deploying a large number of threads and/or tasks in order to extract a significant fraction of the computational power of such hardware.

Current solutions for extracting on-node parallelism are based on operating system (OS) threads in both low- or high-level libraries. Examples of this usage are `Pthreads` [3] for the former and programming models (PMs) such as OpenMP [4] for the latter. However, performing thread management in the OS increases the cost of these operations (e.g. creation, context-switch, or synchronization). As a consequence, leveraging OS threads to exploit a massive degree of hardware parallelism may be inefficient. In response to this problem, dynamic scheduling and lightweight threads (LWTs) (also known as user-level threads, or ULTs) models were first proposed in [5] in order to deal with the required levels of parallelism, offering more efficient management, context switching and synchronization operations. These thread solutions rely on threads that are managed in the user-space so that the OS is only minimally involved and, hence, the overhead is lower.

To illustrate this, Figure 1 highlights the time spent when creating OS thread and user-level threads (ULTs). In this example, one thread is created for each core in a machine with two Intel Xeon E5-2695v4 (2.10 GHz) CPUs and 128 GB of memory. For the OS threads, we employ the GNU C 6.1 library [6], and Argobots (07-2018) threads for the ULT case [7]. The time difference is caused by the implication of the OS and by the features of each type of thread.

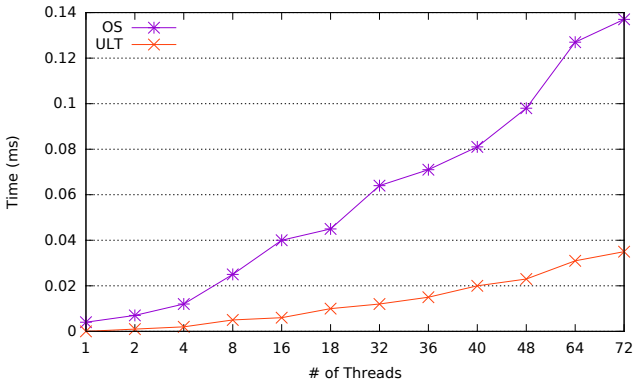

Figure 1: Cost of creating OS threads and ULTs.

For tackling the OS overhead, a number of LWT libraries have been implemented for specific OSs, such as `Windows Fibers` [8] and `Solaris Threads` [9]; for specific hardware such as `TiNy-threads` [10] for the Cyclops64 cellular architecture; or for network services such as `Capriccio` [11]. Other solutions emerged to support specific higher-level PMs. This is the case of `Converse Threads` [12] [13] for Charm++ [14]; and `Nanos++`

LWTs [15] for task parallelism in OmpSs [16]. Moreover, general-purpose solutions have emerged such as `GNU Portable Threads` [17], `StackThreads/PM` [18], `ProtoThreads` [19], `MPC` [20], `MassiveThreads` [21], `Qthreads` [22], and `Argobots` [7]. Other solutions that abstract LWT facilities include `Cilk` [23], `Intel TBB` [24], and `Go` [25]. In addition, solutions like `Stackless Python` [26] and `Protothreads` [19] are more focused on stackless threads.

In spite of their potential performance benefits, none of these LWT software solutions has been significantly adopted to date. The easier code development via directive-based PMs, in combination with the lack of a standard/specification, hinder portability and require a considerable effort to translate code from one library to another. In order to tackle this situation, a common application programming interface (API), called Generic Lightweight Threads (GLT), was presented in [27]. This API unifies LWT solutions under a unique set of semantics, becoming the first step toward a standard/specification. GLT is currently implemented on top of `Qthreads`, `MassiveThreads`, and `Argobots`. One further step is presented in [28] and [29], where we explain the semantical mapping between the OpenMP and OmpSs PMs and LWTs and implement both high-level solutions on top of the GLT API.

In this paper we demonstrate the usability and performance benefits of LWT solutions. We analyze several threading solutions from a semantic point of view, identifying their strong and weak points. Moreover, we offer a detailed performance study using OpenMP because of its position as the *de facto* standard parallel PM for multi/many-core architectures. Our results reveal that the performance of most of the LWT solutions is similar and that these are as efficient as OS threads in some simple scenarios, while outperforming them in many more complex cases.

In our previous work [30], we compared several LWT solutions and used the OpenMP PM as the baseline. In this study we expand that work adding `Pthreads` library to our semantic and functional analysis of threading libraries in order to highlight the overhead (if any) introduced by the OpenMP implementations. The purpose of this paper is to present the first comparison of threading libraries from a semantic point of view, along with a complete performance evaluation that aims to demonstrate that LWTs are a promising replacement for `Pthreads` used both as low-level libraries and as the base implementation for high-level PMs.

The contributions of this paper are as follows: (1) an extensive description of the current and most-used threading solutions; (2) an analysis of their APIs; and (3) a performance analysis designed to illustrate the benefits of leveraging LWTs instead of OS threads.

The rest of the paper is organized as follows.

Section 2 reviews in detail the threading solutions. Section 3 presents an analysis of the different LWT approaches. Section 4 introduces the different parallel patterns that are analyzed. Section 5 provides implementation details on the microbenchmarks we developed for this paper. Section 6 analyzes the performance of LWT libraries. Section 7 briefly reviews related work. Section 8 closes the paper with conclusions.

2. Threading Libraries

In this section we describe the two types of threading libraries, OS threads and LWTs, that are analyzed and evaluated in this paper. Moreover, we briefly present the OpenMP PM, for which production implementations are currently based on `Pthreads`.

For the evaluation of the libraries, from the point of view of OS threads, we have selected `Pthreads` because this is a standard library that matches the current hardware concurrency. In the case of LWTs, `Qthreads` and `MassiveThreads` have been selected because these are among the most used lightweight threading models in high-performance computing (HPC). In addition, `Converse Threads` and `Argobots` were chosen because they correspond to the first (and still currently used) LWT library and the most flexible solution, respectively. Despite `Go` is not HPC-oriented, we have also included it as representative of the high-level abstracted LWT implementations.

Prior to highlight the strengths and weaknesses of each solution, we present a summary of the most used functions when programming with threads. Table 1 lists the nomenclature of each API for different functionality. This includes initialization and finalization functions that set up and destroy the threads environment, as well as the threads/tasklets management (creation, join and yield).

TABLE 1: Summary of the most used functions in microbenchmark implementations using threads. Pth, Arg, Qth, Mth, CTh, and Go identify the threading libraries Pthreads, Argobots, Qthreads, MassiveThreads, Converse Threads, and Go, respectively.

Function	Pth (pthread_)	Arg (ABT_)	Qth (qthreads_)	MTh (myth_)	CTh	Go
Init	–	init	initialize	init	ConverseInit	–
Thread	create	thread_create	fork	create	CthCreate	go
Tasklet	–	task_create	–	–	CmiSyncSend	–
Yield	yield	thread_yield	yield	yield	CthYield	–
Join	join	thread_free	readFF	join	–	channel
End	–	finalize	finalize	fini	ConverseExit	–

2.1. Pthreads API

`Pthreads` [31] offers three PMs that differ in how the threads are bound and which thread is in control. An important agent in these PMs is the *kernel scheduled entity* (KSE). KSEs can be managed directly by the OS kernel and the PM changes depending on the threads–KSE mapping.

The *library–thread model* contains a single KSE, and several threads are scheduled and executed on top of it. This relationship is N:1 and may limit concurrency because a single thread is scheduled at a time. This approach is adopted by the GNU Portable Threads library [32].

The *hybrid model* is composed of a set of KSEs, each managing several threads in an M:N relationship. Since LWT libraries follow this hybrid approach, the `Pthreads` API should be able to accommodate the PM offered by LWTs.

The *kernel–thread model* employs one KSE for each thread that is generated (1:1 relationship).This increases the overhead of the management mechanism because the OS kernel is involved in the scheduling and execution of the

threads. This is probably the most used implementation of the Pthreads API and it is integrated into the GNU C library [6].

Pthreads does not expose KSEs as part of the API, although these are present in its execution model. Hence, the Pthreads implementations interpret KSEs differently, leading to the previously discussed mappings between KSEs and threads (N:1, 1:1, or M:N). Therefore, users do not have control over this mapping; instead, they have to follow the mapping offered by the threading implementation. Although some implementations offer functionality for this mapping (e.g., pthread_setaffinity_np in the GNU C library), this is not supported by the standard and, therefore, changing the underlying Pthreads implementation may produce a misbehavior in the application/runtime.

2.1.1. OpenMP over Pthreads

High-level parallel PMs have been implemented on top of Pthreads in order to promote programming productivity by easing the use of parallel techniques. The most well-known example is OpenMP, an API that supports multi-platform shared-memory multiprocessing programming. Currently, there exist implementations of OpenMP for most platforms, processor architectures, and operating systems. OpenMP exposes a directive-based PM that helps users accelerate their codes exploiting hardware parallelism by adding annotations to the code. At compile time, these annotations are converted to runtime function calls. Intel and GNU provide two commonly used OpenMP implementations that leverage Pthreads in order to exploit concurrency. These runtimes automatically create all the necessary structures and distribute the work among Pthreads.

Since version 3.0, OpenMP supports the concept of tasks, which constitute different pieces of code that may be executed in parallel, and each can be different (e.g., only computation, I/O, communication, etc.). In contrast with work-sharing constructs, distinct OpenMP implementations leverage different mechanisms for task management. In particular, while the GNU version implements a shared task queue for all threads, the Intel implementation incorporates one task queue per thread and integrates an advanced work-stealing procedure for load balancing.

2.2. Converse Threads

Converse Threads [12] [13] was one of the first LWT implementations, developed at the University of Illinois in 1996. It is a parallel language-integration solution designed to enable the interaction of different PMs.

Although Converse Threads was designed and developed more than 20 years ago as a general-purpose solution, it remains in use because it composes the underlying layer of the Charm++ implementation [14]. Since its creation, Converse Threads has been extended with several modules (e.g., client–server) that expand the basic functionality and adapt the PM to diverse application scenarios. This continuous development maintains Converse Threads as an appealing solution for HPC environments.

The Converse Threads PM offers two hierarchical thread levels: process (OS threads) and work units. Processes allocate queues where work units are stored. Users may select the number of active processes by means of environment variables.

As an innovative feature, Converse Threads exposes two types of work units: ULTs and Messages. ULTs are the base of the LWT solutions, and they represent a migratable (a ULT is executed by an OS thread, paused, and resumed by another OS thread), yieldable, and suspendable work unit with its own stack. A "message" represents a piece of code that is executed atomically. Messages lack an own stack and thus cannot be migrated, yielded, or suspended. Instead, these constructs are recommended for inter-ULT communication, for short nonblocking tasks, and as synchronization mechanisms. In addition, only messages can be inserted into other thread's queues and this situation reduces flexibility because some codes (e.g., a blocking code) cannot be executed as a message.

Figure 2 depicts the PM offered by Converse Threads, showing the interaction of Converse Threads processes via messages. In that scenario, Process 0 sends a message to Process 1 that is scheduled and executed. Once Process 1 completes the execution of the ULTs, it communicates to Process 0 the work completion via the insertion of a message into the queue of Process 0.

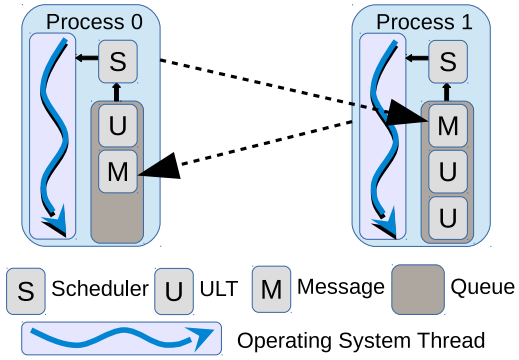

Figure 2: Converse Threads PM and process interaction.

From the point of view of the PM, Converse Threads allows different execution modes, aimed to different scenarios. The behavior is selected with the function ConverseInit, which initializes the environment. If the "normal" mode is selected, threads operate like MPI processes and all the threads execute the overall code. The user is able to select the code portion to execute depending on the thread id. In the "return" mode, Converse Threads creates one thread that acts as the master. This thread dispatchs the work to other threads by sending messages.

The Converse Threads scheduler is a priority system that supports efficiently stackless and standard threads. This scheduler accomodates two strategies: First-In-First-Out (FIFO) and Last-In-First-Out (LIFO). In order to enhance the flexibility of Converse Threads, this library also allows user-defined schedulers that interact with threads.

To complete this concurrent environment, the Converse Threads library offers several concurrent implementations of data structures developed specifically for

this PM, including queues and lists that are automatically managed by the library.

2.3. MassiveThreads

`MassiveThreads` [21] was developed at the University of Tokyo (Japan) in 2014. This LWT library is a recursion-oriented solution that tackles the thread blocking problem when an I/O operation is executed. In addition, this solution provides a tuned load balancing mechanism among threads via work-stealing.

`MassiveThreads` is a consolidated solution in continuous development which can be used in current hardware systems.

As almost every other LWT solution does, `MassiveThreads` offers two hierarchical levels: Workers (the OS thread) and ULTs. Each worker includes its own work unit queue that is managed by a scheduler. The representation of the PM is illustrated in Figures 3 and 4. The default queue scheduler follows the work-first scheduling policy (Figure 3): when a new ULT is created, it is immediately executed, and the current ULT is moved into a ready queue. In this scenario, Worker 0 generates a new ULT and the Main task (labeled as *M*) is moved to the queue. Then, M may be stolen by idle Workers. Although this policy benefits recursive codes, because of the exploitation of data locality, this behavior can be transformed into a help-first policy (Figure 4) at compile time. The help-first policy prevents the worker from executing the new ULTs unless a yield function is called. Therefore, Worker 0 generates a certain number of ULTs that are stored in the queue, and Worker 1 steals the lastly created ULT.

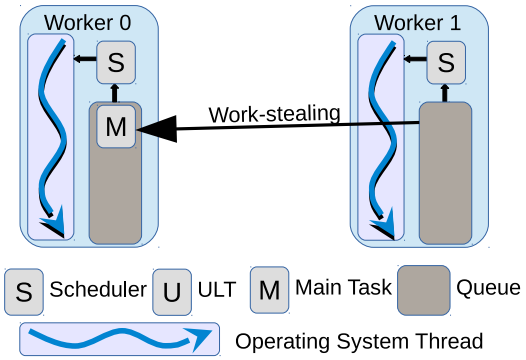

Figure 3: `MassiveThreads` PMs with Work-first policy.

The number of workers that are spawned by the `MassiveThreads` environment is selected by the user via the environment variable `MYTH_NUM_WORKERS`. Once the application is launched, this number cannot be modified.

In contrast with `Converse Threads`, `MassiveThreads` does not allow the introduction of work units into other Workers' queues. Therefore, all the work units are created into the current Worker's queue and the load balance is pursued with a work-stealing mechanism that allows an idle Worker to gain access to the ready queue of other Workers and to steal a ULT from there. The work-stealing mechanism is also depicted in Figures 3 and 4.

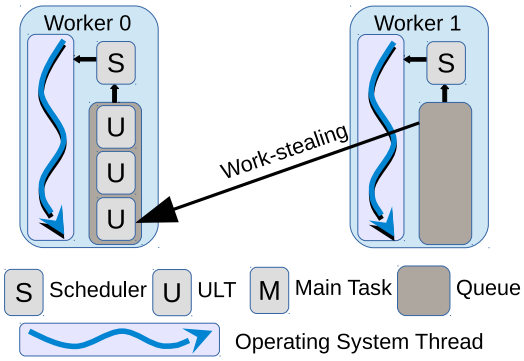

Figure 4: `MassiveThreads` PMs with Help-first policy.

Once the work units are in the queues, the execution follows the LIFO policy for each worker's work, and a FIFO policy in case of work-stealing. This algorithm was selected because this scheduling policy is known to be efficient for recursive task parallelism.

`MassiveThreads` includes a mechanism for I/O handling that consists of three procedures, namely, (1) registering a new file descriptor, (2) performing the I/O call, and (3) polling to resume blocked threads. With this mechanism, `MassiveThreads` tackles the blocking thread problem by overlapping communication and computation.

In order to enhance portability from `Pthreads` to `MassiveThreads`, the latter provides a Pthreads-like API. This feature allows programmers to convert their legacy codes into `MassiveThreads` applications without much effort. Moreover, it allows the use of high-level PMs that are currently written on top of Pthreads, with `MassiveThreads` as the underlying library.

2.4. Qthreads

`Qthreads` [22] was developed at Sandia National Laboratories in 2008 as a general-purpose LWT implementation based on the full/empty bit design. The feature that distinguishes this LWT PM is the use of a hierarchy of three levels instead of the two-level structure of other approaches. The new level is located between the OS thread (called Shepherd) and the work units (ULTs), and it is known as Worker. Shepherds and Workers may be bound to several types of hardware resources (nodes, sockets, cores, or processing units) with the unique restriction that the Shepherd boundary level may lie at a higher level than the Worker level.

Depending on the level of the Shepherds, these may manage one or more Workers. On the one hand, when a Shepherd is bound to a node, it may manage up to *n* Workers, where *n* is the number of logical cores. On the other hand, when a Shepherd is bound to a logical core, it only manages one Worker bound to the same core. These configurations are determined via a few environment variables.

Depending on the number of Shepherds (single or multiple) the user is allowed to select the work unit scheduler during the library configuration step. Figure 5 depicts the `Qthreads` system when one Shepherd is bound to a core and one Worker (omitted for simplicity) is spawned per Shepherd. The scheduler configurations integrate work-stealing in order to attain a fair work-load balance among

Shepherds. In addition, Qthreads enables creating ULTs for specific Shepherds, and those ULTs cannot be stolen by other Shepherds. In Figure 5, Shepherd 1 is not able to steal the last ULT (Assigned ULT), so it steals the previous ULT.

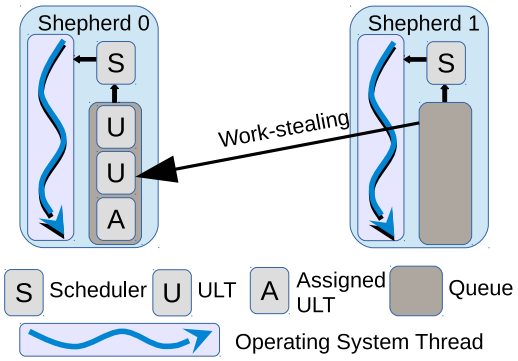

Figure 5: Qthreads PM.

Qthreads allows that a large number of ULTs access any word in memory. Associated full/empty bits are used for synchronization among ULTs as well as for leveraging *mutex* mechanisms. This access to memory requires hidden synchronization, which may severely impact performance.

A large number of distributed structures such as queues, dictionaries, or pools, are offered along with *for loop* and *reduction* functionalities. ULT-aware system call functions are also part of the Qthreads API.

2.5. Argobots

Argobots [7] was developed in 2015 at Argonne National Laboratory. It is presented as a mechanism-oriented LWT solution. This entails that, in addition to being possible to use as a low-level library, it offers the mechanism for building different environments. Therefore, it allows programmers to create their own PM.

Thanks to its development approach, this PM gives the programmer an absolute control over all the resources. In contrast with previous LWT solutions, the OS threads (named Execution Streams, ES) may be dynamically created by the user at runtime instead of at the initialization point with environment variables. Since those ESs are independent, there is no need for an internal synchronization mechanism. Users may also decide the number of required work unit pools, as well as which ESs have access to each pool. These pools may be configured with different access patterns, depending on the number of producers and consumers. For example, a queue may be accessed by a single ES in order to create ULTs, while it may be accessed by several ESs for executing the work units, and vice-versa.

Although a default scheduler is defined for each pool, in Argobots programmers can create their own instances and apply them individually to the desired pools. The default scheduler implements a LIFO policy and only allowed ESs may interact with the scheduler. Furthermore, Argobots supports stackable schedulers, enabling dynamic changes to the scheduling policy that may benefit code portions. The Argobots flexibility is represented in Figure 6. This

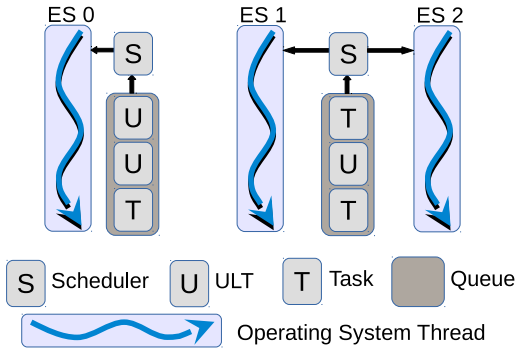

Figure 6: Argobots PM using one private pool for ES 0 and a shared pool for ES 1 and ES 2.

feature enables the programmer to create different environments inside a unique code. As an example, in Figure 6, ES 0 features its own private queue, while ESs 1 and 2 share a work unit queue. This complete flexibility increases programming difficulty but, at the same time, improves code adaptability.

Argobots presents two types of work units: ULTs and Tasklets (as Converse Threads does). The difference is that Tasklets are an atomic piece of work and therefore, it cannot yield, migrate, and/or pause. Argobots low-level API offers a high variety of functionality that enables implementing other LWT solutions and low/high-level runtimes on top of Argobots.

2.6. Go

Go [25], developed by Google in 2009, is an object-oriented programming language focused on concurrency that is practically hidden to programmers. This library abstracts the existence of LWTs from users with the aim of increasing coding productivity in web-service scenarios. This language supports concurrency by means of goroutines which are ULTs executed by the underlying threads. The number of threads can be decided by the user at execution time via the environment variable GOMAXPROCS. In addition, users are able to specify the number of threads for a specific code portion at runtime. Due to the LWT abstraction, Go is the

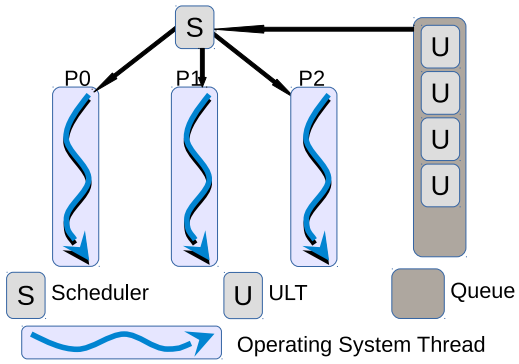

Figure 7: Go PM.

less flexible solution among those reviewed in this paper. The Go mechanisms offered to the programmers include: in the

creation step, the `goroutine` call; and in the joining step, the creation of a communication channel. In the creation step, all threads share a global queue where `goroutines` are stored. This queue is managed by a scheduler that is responsible for assigning the ULTs to idle threads. This global and unique queue needs a synchronization mechanism that may impact performance in case a high number of threads is used. Figure 7 depicts the interactions between `Go` processes and the shared queue.

The synchronization procedure implemented in `Go` is an out–of–order communication channel that may deliver higher performance than the sequential mechanisms. However, it is the programmer's responsibility to identify which `goroutine` has sent the message by checking the returned value.

2.7. Generic Lightweight Threads

GLT [27], developed at Universitat Jaume I de Castelló in 2016, is a generic solution that joins, under a unique PM, the semantics of several widely-used LWT libraries. This is the first effort toward creating a standard in the field of LWTs. The GLT PM is formed by a set of `GLT_threads`, in a number that is specified by means of the environment variable `GLT_NUM_THREADS`. Each `GLT_thread` contains the OS thread, a work unit queue, and a scheduler. As `Argobots` does, GLT allows the use of two types of work units: ULTs and Tasklets (namely `GLT_ult` and `GLT_tasklet`). The GLT scheduling policy is implementation-defined. Therefore, its configurations directly depend on the approaches that are offered by the LWT implementation.

The GLT API is currently implemented on top of `Argobots`, `Qthreads`, and `MassiveThreads`. Therefore, a code written with GLT can be executed by those libraries without any code modification. GLT also implements the `Pthreads` API in order to transparently execute legacy codes over LWTs.

3. Analysis of Threading Libraries

In this section, we first analyze the threading libraries from a semantic point of view, highlighting the most significant features in the field of this type of solutions. Next, we classify the threading implementations from the perspective of ease of use.

3.1. Semantic Analysis

We present a semantical analysis of the threading solutions in order to highlight the different features offered to programmers. Threading solutions were designed to extract the computational power of many-/multi-core architectures. LWT solutions provide a programming model close to that of OS threads. However, since LWTs are executed in the user-space, they avoid som of the overhead experienced by conventional OS threading mechanisms. LWT libraries lie on top of OS threads (OS threads execute the LWTs); however these are created at the beginning of execution and they are not managed by the OS during the application runtime. Although LWT solutions show some common features, each library offers its own functionality and PM.

The most important features of the threading libraries from the PM perspective are summarized in Table 2.

TABLE 2: Summary of the execution and scheduling functionality offered by the LWT libraries. Pth, Arg, Qth, Mth, CTh and Go identify the threading libraries Pthreads, Argobots, Qthreads, MassiveThreads, Converse Threads, and Go, respectively.

Concept	Pth	Arg	Qth	MTh	CTh	Go
# Hierarchy Levels	1	2	3	2	2	2
# work unit Types	1	2	1	1	2	1
Thread Support	✓	✓	✓	✓	✓	✓
Tasklet Support		✓			✓	
Group Control		✓	✓	✓	✓	✓
Yield		✓	✓	✓	✓	
Yield To		✓				
Global work unit Queue	✓	✓				✓
Private work unit Queue	✓	✓	✓	✓	✓	
Plug-in Scheduler	✓	✓	✓	✓	✓	
Stackable Scheduler		✓				
Group Scheduler		✓				

3.1.1. Hierarchical Levels

This number indicates the number of layers inside each PM. Each layer offers its own features and aims for different purposes. The main difference between OS threads and LWTs is that the former only contains one level, `Pthread` itself, while LWTs feature at least two levels. The lowest level in each library is the OS thread representation. In addition, in the case of LWTs, this level usually includes a queue for work units and a scheduler. This object receives different names depending on the library: Execution Stream in `Argobots`, Shepherd in `Qthreads`, Worker in `MassiveThreads`, Processor in `Converse Threads`, and Thread in `Go`. Users may decide the number of elements to spawn via environment variables (Group Control row). This number can also be set by programmers at runtime in the case of `Argobots`. `Pthreads`, schedulers, and queues may be created at runtime by the programmer using the LWT APIs.

`Qthreads` presents an additional level located in between the two already presented. This level is formed by workers that execute the work units and are managed by the Shepherds. This feature enables further hardware adaptability depending on the combination of hardware and applications.

Work units are at the top level of this hierarchy. These are usually ULTs but, in the case of `Argobots` and `Converse Threads`, these may also be a Tasklet. These work units are executed by a OS thread of the lower level or by a Worker in `Qthreads`.

3.1.2. Work Units

The main work units in threading libraries are ULTs in the case of LWT solutions and `Pthreads` in the OS threading implementation. Both types of threads are independent, yieldable and, migratable featuring their own private stack. The main difference is that LWTs are managed in the user space while OS threads are managed by the OS. In addition, `Argobots` and `Converse Threads` support an additional work unit called Tasklet: an atomic piece of code that shares the stack with its executor; in other words, these may be considered as a function pointer. Tasklets (also

known as Tasks) are lighter work units than ULTs and are intended for codes that do not block or context switch.

These work units are stored inside pools/queues waiting to be executed. These structures are also PM-dependent and set the library behavior. In the case of `Argobots` and `Pthreads`, these structures may be created by the user. This increase of difficulty also leads to higher flexibility. Conversely, `Qthreads`, `MassiveThreads`, `Converse Threads` and `Go` hide that feature to users, and the environment may be modified only via environment variables or at compilation time. `Go` only allows the use of one shared pool while `Qthreads`, `MassiveThreads`, and `Converse Threads` by default assign one queue/pool per OS thread.

3.1.3. ULT Management

Another feature that defines the PM is the functionality that a programmer is allowed to use over the work units. `Go` is the most restrictive solution. This implementation does not include basic threading operations (such as yield, cancel, or resume) which are exposed by other libraries. Moreover, only one shared queue is employed, and hence the scheduling options are reduced. At the other extreme, `Argobots` includes not only the common functionality but also an improvement of the yield call. The `yield_to` function allows the programmer to pause a current work unit and start (or resume) another ULT without asking the scheduler. In addition, `Argobots` enables programmers to create their own environment.

All these features yield `Argobots` as the most flexible solution. The other solutions present default schedulers and environment configurations. As a novelty, `Qthreads` enables programmers to create work units for a specific Shepherd. In the case of `MassiveThreads`, it only creates work units inside the current Worker's queue and an internal work-stealing mechanism ensures the load balance. Although some `Pthreads` implementations allow the use of yield functions, this functionality is not included in the API specification. To identify this, the corresponding API function names are appended with the `_np` suffix that means "non-portable".

3.2. Usage Difficulty Analysis

Although semantics are important when selecting a library for generating a code solution, the adoption of a PM is also dependent on its ease of use. From this perspective, we classify the presented libraries into three levels depending on the features that are offered to the programmers for building their application/PM environment.

In the **easy** level, we include `Go` and `MassiveThreads`, because of the reduced number of features that depend on the programmer. In the former, the user is responsible of selecting the number of OS threads and then creating and joining the ULTs. In the latter, the programmer also needs to allocate the resources and select whether the scheduler may use a Work-first or a Help-first strategy.

In the **medium** level we find `Qthreads`. In this solution the programmer is involved in more decisions which may affect performance. The users may select a combination of the number of Shepherds and Workers via environment variables. In addition, the users may decide the boundaries of each element (e.g., node, socket, core, or processing unit). Once the environment is set, at the coding level the programmer decides whether the ULTs is bound or not to a specific Shepherd.

In the **difficult** level we include the three remaining solutions: `Pthreads`, `Converse Threads`, and `Argobots`, though for different reasons. While not being a LWT solution, `Pthreads` enforces the use of low-level functionality if the programmer wants to generate an environment where `Pthreads` share structures and interact among them. `Converse Threads` offers three distinct models of execution and each model follows its own rules. This feature forces programmers to understand deeply each one of the models in order to select the most appropriate in each case. For all scenarios, programmers may manage the ULTs in addition to the Tasklets (or messages) for communication among Processors. `Argobots` offers complete flexibility for environment generation. In addition, this environment can be changed at run time so one part of the application can behave totally different from others. In this LWT library the programmer indicates the number of Execution Streams, the number of pools, the relationship among Execution Streams and pools, which scheduler policy follows each pool, etc. As usual, increasing the flexibility in the library implies more control yet also more work from the programmers.

4. Parallel Code Patterns

Many scientific applications may benefit from the use of OpenMP in order to shorten their execution time. The basic mechanism consists of using OpenMP *pragmas* in order to hint the compiler the code that can be executed concurrently. The compiler translates these directives into OpenMP runtime calls and, at run time, the code is executed in parallel. In this section, we present the most common parallel patterns and explain how current OpenMP runtimes convert the pragmas into parallel code. These patterns will be the basis for our performance evaluation of the different threading solutions.

4.1. For Loop

The most frequently used OpenMP directive and probably also the shortest path to produce parallel code is `#pragma omp parallel for`. It can be placed right before a parallel loop composed of independent iterations and produces code where each thread executes a subset of the iteration space. This thread management is transparent to programmers, who are in charge of annotating the parallelizable code with the *pragma*.

Widely-used OpenMP runtimes, such as `gcc` and `icc`, handle this scenario similarly. The master thread sets a pointer to a function call containing the parallel code in each thread data structure and it is also responsible for calling the function. Besides, all threads wait in a barrier (unless a `nowait` clause is present) at the end of the parallelized loop.

In the case of LWT solutions, the main thread generates one work unit per thread and divides the number of iterations among them. This work unit contains a function pointer to be executed. All the arguments are passed via a structure that contains critical information (e.g., the number of iterations,

```
1  void for_lwt(void * args)
2  {
3    arg_for *arg = (arg_for*) args;
4    for (int i = arg.ini; i < arg.fini; i++)
5      code(i);
6  }
7    ...
8    //Main function
9    //Allocate memory for structures
10   ULT * lwts[NUM_ULTS]; arg_for * args[NUM_ULTS];
11
12   for (int i = 0; i < NUM_ULTS; i++)
13   {
14     //Calculate the number of iterations per LWT
15     ...
16     //Arguments initialization
17     args[i].ini = XXX;  args[i].fini = XXX;
18     //LWT creation
19     create_lwt(for_lwt, args[i], lwts[i]);
20   }
21
22   lwt_yield();
23
24   //Wait for LWT completion
25   for (int i = 0; i < NUM_ULTS; i++)
26     join_lwt(lwts[i]);
```

Listing 1: OpenMP `for` loop parallelism implemented with LWT solutions.

variables, etc.) that is necessary to execute the function. Listing 1 shows an abstraction of how the OpenMP `for` loop *pragma* with a static scheduler is translated into LWT code. Lines 1–6 compose the function that is executed inside the ULTs. Lines 8–26 are part of the main function where the data structures for ULTs and arguments are initially allocated (Line 10); Lines 12–20 correspond to the division of the number of iterations among the number of ULTs, the argument initialization, and the ULT creation (line 19). Line 22 allows the main thread to call the scheduler and execute a ready ULT. Once the control is returned to the main function, all the ULTs are joined (line 26) and the work is completed. This example highlights the complexity of the code utilizing low-level LWT APIs.

4.2. Task Parallelism

Task parallelism appeared in the OpenMP 3.0 specification as a solution to parallelize unbounded loops and recursive codes. Its usage, however, has been spread to all types of applications that contain pieces of code that can be executed in parallel or that present dependencies among them. In the second case, the runtime generates a directed acyclic graph of tasks and their dependencies. OpenMP tasking follows the LWT approach in the sense that tasks are pieces of enqueued code waiting to be executed by an idle thread. This is expressed with the pragma `#pragma omp task`. However, different OpenMP runtimes leverage their own internal approach for tasking. More concretely, `gcc` OpenMP creates a shared queue to store tasks. This queue is accessed by all the team's threads. In contrast, `icc` allocates one private queue for each thread in the team. This implementation reduces the contention generated by a shared queue. It also implements a work-stealing mechanism for load-balancing purposes. This mechanism is used when a thread's task queue is empty and the thread becomes idle. Storing a high number of tasks can reduce performance because of contention and the cost of queue reallocation; therefore, `gcc` and `icc` include a non-configurable cutoff mechanism. Once a specific number of tasks are stored (64 times the number of threads for `gcc` and 256 in a thread's queue in the case of `icc`), the new non-dependent tasks avoid the queues and are executed immediately. The situations described in the following two subsections can occur, depending on the code that creates the tasks.

4.2.1. Single Region

In this scenario, a thread executing a `single` or `master` OpenMP region (`#pragma omp single` or `#pragma omp master`) creates all the tasks in that region. Meanwhile, the other threads in the team execute them. Once the thread that creates the tasks completes its work, it joins the others in the task execution process.

The OpenMP implementation in `gcc` uses one shared queue for all threads and the created tasks are pushed into that queue. The threads in the team compete for access to a task. The protection of the queue is enforced via *mutex* and thus contention can increase with the number of threads. Conversely, `icc` OpenMP uses one private queue for each thread. This situation triggers the work-stealing because the other threads are idle. The performance in this scenario is also affected by contention because all threads are trying to gain access to the queue. The effectiveness of the work-stealing mechanism may well affect performance.

When using the threading solutions, the main thread generates one work unit per OpenMP task and, as in the `for` loop scenario, the work unit is created with the function pointer and the necessary data.

4.2.2. Parallel Region

This scenario occurs when all the threads in a team execute parallel code that creates tasks. During execution, the threads push the new tasks into the task queue (if the cutoff value is not reached), and once this is done, the threads execute the tasks. For `gcc` OpenMP, threads compete to gain access to the shared queue twice: the first time in order to create the tasks and the second time to obtain tasks to execute. In `icc` OpenMP, each thread generates the tasks into its own queue. With this approach, work-stealing is almost nonexistent thanks to a better load balance.

When threading libraries are employed, we have two different parallel levels. The first is mapped to the parallel region, where the main thread generates a work unit for each thread with the function pointer of the region. Then, each thread executes the parallel region creating its own work units which are the OpenMP tasks.

4.3. Nested Parallel Constructs

When one or more parallel *pragmas* are found inside a parallel code, we have nested parallelism. In this case, for the first *pragma*, the runtime spawns a team of threads and. for the second pragma, each thread in the team becomes the master thread of its new threads' team. In this scenario, the number of active threads grows quadratically. Nested parallelism is not frequent in current applications because performance drops when the number of threads exceeds the number of cores (oversubscription). However, this can still

occur in some situations that the user may be unaware of. For example, a programmer accelerates the code with OpenMP *pragmas* and, in this concurrent code, threads invoke external library functions that are parallelized using also OpenMP *pragmas*.

The way OpenMP solutions manage nested parallelism differs. The `icc` OpenMP runtime fulfills the new thread teams reusing idle threads (if any) or creating new threads. Conversely, `gcc` OpenMP does not reuse the idle threads; therefore, each time an OpenMP pragma is executed, a new team of threads is created. Since the idle threads are not deleted (but stored in a thread pool), the number of active threads in the system may increase exponentially. To reproduce this scenario, we have implemented a code consisting of two nested `for` loops, each with its own `#pragma omp parallel for` directive.

With LWT libraries, the outer `for` loop implementation follows the behavior of that pattern and each work unit executes a range of iterations of the outer loop. Then each work unit creates as many work units as number of threads and divide the iterations of the inner loop among these.

5. Microbenchmark Implementation Details

In this section, we discuss how we adapted our microbenchmarks implementing the patterns described in Section 4 to the specifics of each LWT library. [1]

5.1. Pthreads

Although `Pthreads` is the supporting library used in production OpenMP and other high-level PMs, we also used this solution as a low-level threading option for our microbenchmarks. In those cases where OpenMP tasks are employed, we generate one `Pthreads` per task and limit the available number of cores by means of the `taskset` command. The thread management relies on the OS itself for all the microbenchmarks.

5.2. Converse Threads

For `Converse Threads`, we employ the "return" mode and leverage Messages. The former enables us to follow the OpenMP approach, where there is one master thread while the other processes are treated as slaves. The usage of Messages is necessary because this is the only type of work unit that can be pushed into other threads' queues, and therefore the only way to mimic OpenMP's behavior. In this scenario, the master thread creates as many Messages as threads and pushes them onto other threads' queues. This limitation, however, prevents the use of `Converse Threads` to support parallel codes because messages cannot yield and hence the requirements of OpenMP are not fulfilled.

5.3. MassiveThreads

We have analyzed both Work-first and Help-first policies. However, only the best of these is shown in Section 6. The difference among these two policies lies on the way a new work unit is treated. While the former pushes the current work unit into the queue and executes the recently created work unit, the latter pushes new work units into the queues while the current task continues its execution.

1. Available at https://github.com/adcastel/ULT_work/tree/master/lwt_microbenchmarks

5.4. Qthreads

With its three levels of hierarchy, `Qthreads` accommodates multiple possibilities in order to attain high performance in a variety of situations. We have tested a set of combinations, including one Shepherd managing the complete node (it manages up to 72 Workers), one Shepherd per socket (each manages up to 36 Workers), and one Shepherd per core (each manages just one Worker). After a preliminary analysis, we chose two combinations: one Shepherd bound to a node and one Shepherd per core. The first choice is more efficient when there is a reduced number of work units, at the cost of increasing the load imbalance; the second option is more appropriate for scenarios with a higher number of work units. In the presentation of our results we discarded the option with a single Shepherd per socket because it offered lower performance than the other choices for all scenarios.

We also test the functions `qthread_fork` and `qthread_fork_to`, which differ in the work queue where the new work unit is stored. While the former pushes the work unit into the current Shepherd's queue, the latter pushes the work unit into a different Shepherd's queue. If `qthread_fork_to` is chosen, the main thread distributes the work using a round-robin dispatch. Hence, four implementations have been evaluated for each test: `qthread_fork` with one Shepherd per node, `qthread_fork` with one Shepherd per core, `qthread_fork_to` with one Shepherd per node, and `qthread_fork_to` with one Shepherd per core.

5.5. Argobots

The flexibility offered by `Argobots` is two-fold. On the one hand, two different types of work units can be used: ULTs and Tasklets. On the other hand, the work unit pools can be private for each thread or shared among all of them. If the private pool option is selected, the main thread needs to dispatch the work units directly to each thread's pool in a round-robin fashion. Therefore, four possible implementations have been tested. Since Tasklet does not have its own stack and is not yieldable, in those scenarios that require two steps of parallelism (nested and task parallelism), the first of them is performed using ULTs.

5.6. Go

This library enables only one implementation due to its unique shared work unit queue. All work units need to be pushed into this queue, as the `gcc` OpenMP task implementation does. Therefore, only one possibility is analyzed.

6. Performance Evaluation

In this section, we first review the work dispatch/synchronization in both threading solutions and the OpenMP PM. Next, we analyze the different parallel code patterns presented in Section 4 (see implementation details in Section 5). The experiments were performed on an Intel 36-core (72 hardware threads) server composed by two Intel Xeon E5-2695v4 (2.10 GHz) CPUs and 128 GB of memory. GNU's `gcc` 6.1 compiler was used to compile the LWT libraries and OpenMP examples. Intel `icc`

compiler 17.0.1 was used to evaluate the performance of the OpenMP implementations, linked with the OpenMP Intel Runtime 20160808 version. For LWT libraries, we employed `Argobots`, `Converse Threads`, and `Go` libraries updated to 07-2018, `Qthreads` 1.10, and `MassiveThreads` 0.95. All results are the average of 500 executions. The maximum relative standard deviation observed in each experiment is between 2% and 5%. Please, note that the microbenchmarks' codes are well-balanced so all OpenMP solutions may offer their best performance.

6.1. Basic Functionality

In the OpenMP implementations, a master thread (or work unit) is in charge of creating secondary threads (or work units), and then distributing the work among these slaves. Once this is done, the master completes its work and waits for the synchronization that indicates that the overall work is completed. This completion can be enforced using different mechanisms, such as barriers, messages, or thread joins.

Although parallel codes may vary depending on different features, such as granularity, the type of code, or the data locality, the work dispatch and join steps are clearly critical for performance, especially in fine-grained codes.

Figure 8 shows the overhead in terms of time spent creating a single work unit for each thread. In this scenario, the main thread creates and dispatches the work units. As expected, increasing the number of created work units increases the execution time. As an exception, `MassiveThreads` (labeled as MTH) maintains the performance because the new work units are created into the master's own queue. Intel and GNU OpenMP runtimes, labeled as `ICC` and `GCC`, follow the trend of LWT solutions. `Go`'s performance corresponds to the usage of a single shared queue, and therefore, contention is added when the number of threads is increased.

`Converse Threads` and `Argobots` Tasklet, labeled as CTH and ABT(T), employ the tasklets to yield the best performance, thanks to its nature, offering slightly higher performance than `Argobots` when ULTs (ABT(U)) are used and being two times faster than the `Qthreads` (QTH) implementation. As expected, the results show that creating `Pthreads` (PTH) is more expensive than creating LWTs (excluding the `Go` implementation) because the OS is responsible for managing the creation. The difference between the `Pthreads` and OpenMP implementations appears when all threads are created in a previous parallel section (in OpenMP), and hence the time spent corresponds only to the cost of performing the actual work.

Figure 9 displays the time required by the master thread to complete the distinct implementations of the joining mechanism. More concretely, `gcc` OpenMP and `Converse Threads` (labeled as GCC and CTH, respectively) employ a barrier which results in an increment of time with the number of threads. Although `Converse Threads` uses Tasklets, this mechanism does not benefit from it. The marked increase of time in `icc` OpenMP is due to the use of more than one thread per CPU. The master thread accesses memory allocated by other threads and therefore the overhead is increased. The remaining libraries use a join

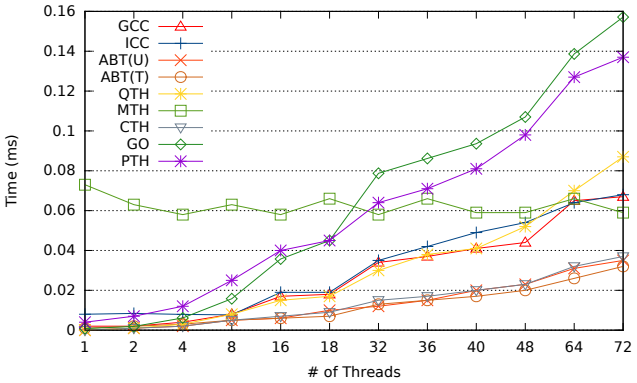

Figure 8: Time of creating one work unit for each thread.

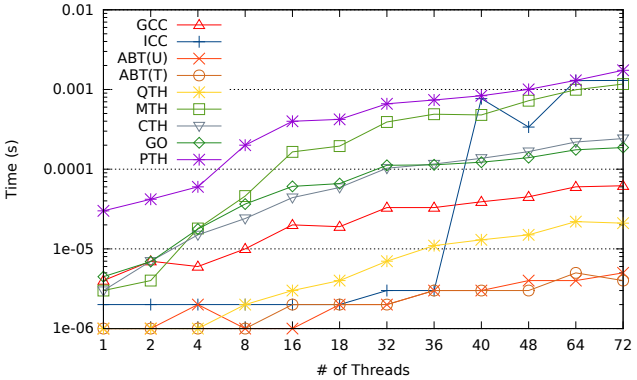

Figure 9: Time of joining one work unit for each thread.

mechanism; while `Go` implements an out–of–order channel communication, `Qthreads` and `Argobots` follow a sequential algorithm that checks the status of the memory word or work unit, respectively. The main difference is that `Argobots` checks the status and deallocates the work unit structure. `MassiveThreads` and `Pthreads` deliver the lowest performance. In the former case, since the main task can be executed by any Worker, each time a thread is joined, several checks are triggered. In the latter case, the OS itself waits until the thread has finished its work and frees the allocated memory.

6.2. Code Patterns

In order to maintain a fair comparison among patterns, and at the same time, avoid code modification, we have selected a BLAS-1 function that matches perfectly the fine-grained approach of LWTs and is highly parallelizable. We implement a `scal` function, which multiplies (and overwrites) the components of a vector by a scalar. We avoid coarse-grained codes because in those, the thread management overhead is totally hidden by the execution time.

In the scenarios where loops (`for` loop and nested `for` loop) are employed, the iterations are divided among the number of threads. In the task-parallel cases, one task is created for each vector element attaining a markedly fine-grained code. This extreme fine level of granularity is chosen in order to understand the behavior of each LWT solution, because this type of parallelism does not hide the thread management overhead. Conversely, if the execution time of

a piece of code is sufficiently long, the overhead is hidden, and therefore there is no significant performance difference between using LWTs or OS threads.

6.3. For Loop

For this test, we have created a vector with 1,000 elements that results in a 1,000-iteration `for` loop. Figure 10 illustrates the results. The implementations used in this experiment are `Argobots` with private pools, `Qthreads` using a single Shepherd per CPU, and the `MassiveThreads` Help-first policy, because these attain higher performance than the alternative configurations for each LWT solution. While `Argobots` (ABT(T) and ABT(U)) presents the highest performance thanks to its low creation and join times (see Figures 8 and 9), the alternative solutions experiment a notable overhead when increasing the number of threads. `Qthreads` (QTH) shows a low execution time because of its small join time, but this behavior changes when using more Shepherds than the number of cores (36). Once this number is reached, the total time constantly increases because of the resource sharing overheads. `MassiveThreads`, `Pthreads`, and `Converse Threads` (labeled MTH, PTH, and CTH, respectively) are 25 times slower than `Argobots` or `icc`. `MassiveThreads` suffers from work-stealing and `Pthreads` from OS management overheads. When `Converse Threads` uses more threads than physical cores, the performance drops due to synchronization. Although `Go` suffers from contention due to the shared queue, its performance is close to the HPC-oriented solutions thanks to its small overhead in the joining mechanism. The performance of the OpenMP Intel and GNU implementations (labeled as ICC and GCC) is close to that of `Argobots` and `Qthreads`, respectively.

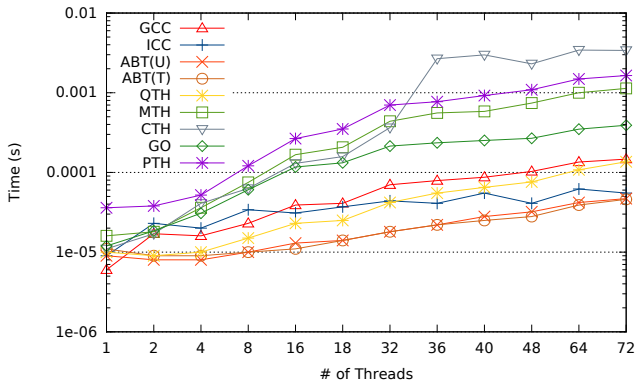

Figure 10: Execution time of 1,000-iteration `for` loop.

6.4. Task Parallelism

For task parallelism, we also used a 1,000-element vector, creating a task per vector element. Figure 11 exposes the execution time when the 1,000 tasks are created by a single thread in a single region. In this case, the LWT implementations correspond to `Argobots` with one private pool per thread, `Qthreads` using one Shepherd per core, and `MassiveThreads` with the Work-first scheduler policy. The OpenMP environment has been modified by setting the OMP_WAIT_POLICY variable to `passive` in order to decrease the overhead caused by the contention in the task queue. In this scenario, both `Argobots` work units (Tasklets and ULTs) obtain the highest performance. The reason for this resides on its lighter management mechanisms and its ES independence, which avoid internal synchronization procedures. The elevate number of work units increases the difference between `Argobots` ULTs and Tasklets. This demonstrates that when the code does not need any context switch (e.g. blocking call, communication, I/O), the use of Tasklets improves performance. `Argobots` Tasklets are a copy of `Converse Threads` Messages and hence the proximity of their results. Both stackless units reduce the execution time by a factor of two compared with ULT implementations.

`Converse Threads` attains one of the highest performances (up to 12 times faster than `Pthreads`) thanks to its messages and management, which are lighter than the ULT functions. `Qthreads` performs slightly lower (2.5 times) than the previous solutions because of two reasons: the use of full-empty bit checks in each memory word and the utilization of more Shepherds than physical cores, which requires additional synchronization. `Go`, `icc`, and `gcc`'s performance lie in the middle. This situation demonstrates that the use of an elevated number of tasks negatively affects those solutions. The `icc` results reveal the effects of the work-stealing mechanism. Workers accessing the master thread's queue to steal work units add contention. `Go` behaves similarly to `gcc` because both rely on a single shared queue. The lowest performance is attained by `Pthreads` and `MassiveThreads`: the former because we create 1,000 OS threads, which causes severe oversubscription; and the latter because the work-first policy implies that, each time a new task is created, the main task is stolen by another thread. As a result, data locality is reduced and there is a drop in the overall performance.

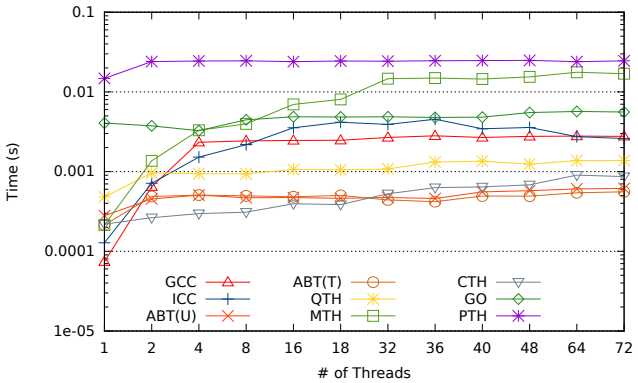

Figure 11: Execution time of 1,000 tasks created into a single region.

Another tasking scenario takes place when tasks are created inside a parallel region. In that case, each thread creates its own work units. This situation is a two-step algorithm. In the first step, as in the `for` loop scenario, the iterations are divided among the threads; in the second step, the tasks are created.

In this scenario, the choices for each threading solution are the same as in the previous test. Figure 12 displays the

results for this experiment. The use of the two-step algorithm affects negatively Go and Converse Threads. Go suffers from contention added by the shared queue, whereas the synchronization mechanism in Converse Threads represents more than 70% of the total execution time. The main reason is that Converse Threads needs additional yield calls due to the use of Messages for the first step. MassiveThreads is more efficient in this case, because this library designed with for recursivity in mind. In addition, all the threads in MassiveThreads are busy, so the work stealing is almost non-existing. Qthreads performance is affected negatively by adding more threads and becomes much slower than other ULT libraries (up to 32 times slower than Argobots). Almost all the time difference is due to the join mechanism. Although both Argobots implementations use ULTs (that can yield) in the first step, the difference between ULTs and Tasklets is negligible.

On the OS threads side, icc offers the highest performance because work stealing has disappeared. This is caused by a perfect load balance. gcc outperforms other solutions thanks to its cut-off mechanism (up to eight threads) and to the wait policy value set as in the previous test, attaining results similar to those of Qthreads. The lowest performance comes again from the Pthreads solution due to the oversubscription caused by the creation of 1,000 threads.

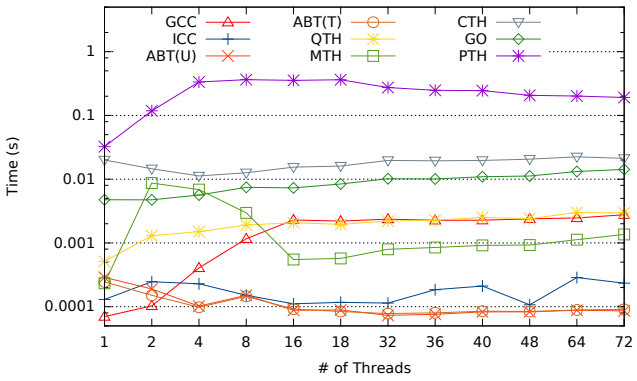

Figure 12: Execution time of 1,000 tasks created into a parallel region.

6.5. Nested Parallel Structures

In this case we have implemented a microbenchmark composed by two nested 1,000-iteration for loops. The choices for LWT libraries are Argobots using a private pool for each thread, Qthreads employing one Shepherd per Worker, and MassiveThreads with Work-first policy. Figure 13 shows the results for this test. The trends shown by both OS-based approaches, OpenMP and Pthreads, are different in comparison with that of LWT libraries shown in previous results. This behavior is caused by the suboptimal implementation of the nested parallel structures in the case of OpenMP and oversubscription in the case of Pthreads. The gcc OpenMP creates new threads for each nested *pragma* directive and avoids reusing idle threads. As a result, when executing this case with 36 threads, the gcc OpenMP spawns 35,036 threads (36 threads for the main team, and 35 threads more for each outer loop iteration). In

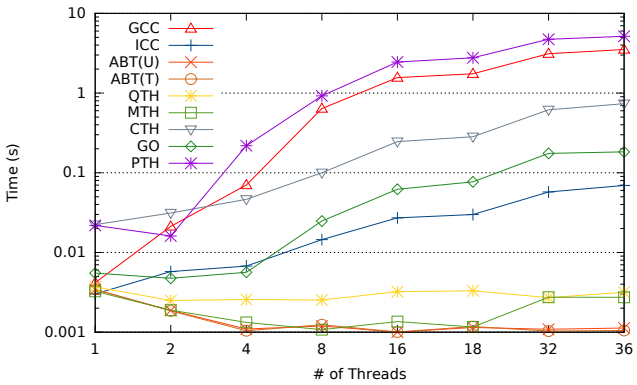

Figure 13: Execution time of a nested parallel for structure with 1,000 iterations per loop.

contrast, the icc OpenMP makes use of the idle threads. However, this approach does present the creation of large number of threads. Concretely, it creates 1,296 threads (36 threads for the main team, and 35 for each secondary team). This number is considerably higher than the total number of cores (72), causing severe oversubscription. The Pthreads implementation performs close to the gcc solution because it follows the same approach. As in previous tests that follow a two step algorithm, Go and Converse Threads offer low performance. Go suffers the contention caused by the use of a single shared queue. In the case of Converse Threads, the addition of yield and barrier functions slow the execution. However, these still perform better than the gcc and Pthreads implementations. The three general-purpose solutions (Argobots Tasklets/ULTs, Qthreads, and MassiveThreads) avoid oversubscription by creating work units instead of OS threads and hence these yield the highest performance. Avoiding the oversubscription problem reduces the OS thread management overhead, increasing performance with respect to the Intel OpenMP approach by factors of 62, 21, and 25 for Argobots, Qthreads, and MassiveThreads, respectively.

7. Related Work

Several studies have been carried among the past years in order to accomplish two goals: show people that using LWT can improve the current performance of parallel hardware via reducing the OS threads overhead, and present a new LWT solution that improves the already existent libraries. Although concurrent multicore hardware is relatively "new", the concept of LWT was introduced in [5]. That work sets the foundations of LWT solutions such as scheduling and management. Converse Threads was later presented in [12] as one of the first low-level LWT libraries. In [22] Qthreads was presented and compared against the Pthreads library via a set of microbenchmarks and applications. MassiveThreads was presented in [21] which also included a performance comparison among MassiveThreads, Qthreads, Nanos++, Cilk, and Intel TBB on several benchmarks. The work in [7] describes the internals of Argobots evaluates its performance against Qthreads and MassiveThreads using microbenchmarks and applications. The Go environment and its procedures are presented in [25]. Generic Lightweight

Threads (GLT) was presented in [27], with the goal of unifying LWT solutions under a single set of semantics. An analogue ULT programming model is the task-based oriented solutions such as CompSs [33], OmpSs [16], `Intel TBB` [24] or a fine-grained OpenMP task employment [34]. These approximations hide the LWT environment by adding an upper layer that abstracts the LWT mechanisms.

8. Conclusions

We have presented a complete analysis of a set of threading solutions including both OS threads and LWTs. In addition, we have performed a PM decomposition of the threading libraries indicating their features. Finally, we have proved, by means of experimental tests, that LWTs provide an appropiate solution for fine-grained parallel codes. For that purpose, we have implemented the most common OpenMP parallel patterns on top of different LWT libraries, showing these offer a performance that is, at least, as good as that attained with `Pthreads` and the OpenMP runtimes. In the case of Intel OpenMP, we have identified some design aspects that can lower the performance of common user code patterns. However, current OpenMP solutions were designed for the old concurrent hardware and it is difficult to migrate from OS threads to other approaches. These issues may limit its usability in near-future applications.

In summary, using LWTs in OpenMP critical patterns, such as task or nested parallelism, can improve the performance compared with the most-used OpenMP implementations, especially in scenarios when it is crucial to extract all the computational power of exascale systems.

Acknowledgments

The researchers from the Universitat Jaume I and Universitat Politècnica de València were supported by project TIN2014-53495-R of the MINECO and FEDER, and the Generalitat Valenciana fellowship programme Vali+d 2015. Antonio J. Peña is financed by the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant No. 749516. This work was partially supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

[1] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The Sunway TaihuLight supercomputer: System and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[2] "TOP500 Supercomputer Sites," http://www.top500.org/.

[3] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[4] "OpenMP 4.5 specification," www.openmp.org/.

[5] D. Stein and D. Shah, "Implementing lightweight threads," in *USENIX Summer*, 1992.

[6] "GNU C Library," www.gnu.org/software/libc/, Accessed July 2016.

[7] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, S. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[8] Microsoft MSDN Library, "Fibers," https://msdn.microsoft.com/en-us/library/ms682661.aspx.

[9] "Programming with Solaris Threads," https://docs.oracle.com/.

[10] J. d. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "TiNy threads: A thread virtual machine for the Cyclops64 cellular architecture," in *Fifth Workshop on Massively Parallel Processing*, April 2005.

[11] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for Internet services," in *Proc. 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 268–281.

[12] L. V. Kale, M. A. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An interoperable framework for parallel programming," in *the 10th Int. Parallel Processing Symposium*, 1996, pp. 212–217.

[13] L. V. Kale, J. Yelon, and T. Knuff, "Threads for interoperable parallel programming," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996, pp. 534–552.

[14] L. V. Kale and S. Krishnan, *CHARM++: A portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.

[15] BSC, "Nanos++," https://pm.bsc.es/projects/nanox/.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[17] "GNU Pth - The GNU Portable Threads," http://www.gnu.org.

[18] K. Taura, K. Tabata, and A. Yonezawa, "StackThreads/MP: Integrating futures into calling standards," in *the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999, pp. 60–71.

[19] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, 2006, pp. 29–42.

[20] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *14th International Euro-Par Conference on Parallel Processing*, 2008, pp. 78–88.

[21] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*. Springer Berlin Heidelberg, 2014, vol. 8665, pp. 222–238.

[22] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *the Workshop on Multithreaded Architectures and Applications*, April 2008.

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[24] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[25] F. Schmager, N. Cameron, and J. Noble, "Gohotdraw: Evaluating the go programming language with design patterns," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 10.

[26] "Stackless Python," http://www.stackless.com.

[27] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, "GLT: A unified API for lightweight thread libraries," in *Int. European Conf. on Parallel and Distributed Computing*, Spain, 2017.

[28] A. Castelló, R. Mayo, K. Sala, V. Beltran, P. Balaji, and A. J. Peña, "On the adequacy of lightweight thread approaches for high-level parallel programming models," *Future Generation Computer Systems*, vol. 84, pp. 22–31, 2018.

[29] A. Castelló, S. Seo, R. Mayo, P. Balaji, E. S. Quintana-Ortí, and A. J. Peña, "GLTO: On the adequacy of lightweight thread approaches for OpenMP implementations," in *Proceedings of the International Conference on Parallel Processing*, Bristol, UK, August 2017.

[30] A. Castelló, A. J. Peña, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Ortí, "A review of lightweight thread approaches for high performance computing," in *Proceedings of the IEEE International Conference on Cluster Computing*, Taipei, Taiwan, September 2016.

[31] "Pthreads API," http://www.cs.wm.edu/wmpthreads.html.

[32] "GNU Portable Threads," www.gnu.org, Accessed July 2016.

[33] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3, pp. 32–36, 2015.

[34] G. Tagliavini, D. Cesarini, and A. Marongiu, "Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight openmp tasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2150–2163, 2018.

Adrián Castelló received his BS degree in computer science, the MS degree in advanced computer systems, and his Ph.D. degree in Computer Science from Universitat Jaume I in 2011, 2013 and 2018, respectively. He is a Post-doc researcher at the same university and his research interests include deep neural networks, programming models and distributed and shared memory systems.

Rafael Mayo received the BS degree from Polytechnic Valencia University in 1991. He obtained his PhD in Computer Science in 2001 at the same University. Since October 2002 he has been an Associate Professor in the department of Computer Science and Engineering in the University Jaume I. Nowadays he is involved in several research efforts on HPC energy-aware systems, cloud computing and HPC system and development tools.

Sangmin Seo is a software engineer at Ground X. Prior to that, he worked as an assistant computer scientist at Argonne National Laboratory and a senior engineer at Samsung Research. He received the B.S. degree in computer science and engineering and the Ph.D. degree in electrical engineering and computer science from Seoul National University, respectively. His research interests include high-performance computing, parallel programming models, compilers, runtime systems, and blockchains.

Pavan Balaji holds appointments as a Computer Scientist and Group Lead at the Argonne National Laboratory, as an Institute Fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University, and as a Research Fellow of the Computation Institute at the University of Chicago. He leads the Programming Models and Runtime Systems group and his research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, and modern system architecture.

Enrique S. Quintana-Ortí received the bachelor and Ph.D. degrees in computer sciences from the Universidad Politecnica de Valencia, Spain, in 1992 and 1996, respectively. He is a Professor in Computer Architecture in the Universitat Poitècnica de València. Recently, he has participated/-participates in EU projects such as TEXT, INTERTWinE, EXA2GREEN and OPRECOMP. His current research interests include parallel programming, linear algebra, energy consumption, transprecision computing and bioinformatics as well as advanced architectures and hardware accelerators.

Antonio J. Peña holds a BS+MS degree in Computer Engineering (2006), and MS and PhD degrees in Advanced Computer Systems (2010, 2013), from Universitat Jaume I, Spain. He is currently a Sr. Researcher at Barcelona Supercomputing Center (BSC), Computer Sciences Department. Antonio works within the Programming Models group where he leads the "Accelerators and Communications for HPC" team. Dr. Peña is also the Manager of the BSC/UPC NVIDIA GPU Center of Excellence. He is a Marie Sklodowska-Curie Fellow.